Using a Class Abstraction Technique to Predict Faults in OO Classes

A case study through six releases of the Eclipse JDT

Djuradj Babich and Peter J. Clarke
School of Computing
and Information Sciences
Florida International University
Miami, FL 33199, USA
{dbabi001, clarkep}@cis.fiu.edu

James F. Power
Department of
Computer Science
National University of Ireland
Maynooth, Co. Kildare
jpower@cs.nuim.ie

B. M. Golam Kibria
Department of
Mathematics & Statistics
Florida International University
Miami, FL 33199, USA
kibriag@fiu.edu

ABSTRACT

In this paper, we propose an innovative suite of metrics based on a class abstraction that uses a taxonomy for OO classes (CAT) to capture aspects of software complexity through combinations of class characteristics. We empirically validate their ability to predict fault prone classes using fault data for six versions of the Java-based open-source Eclipse Integrated Development Environment. We conclude that this proposed CAT metric suite, even though it treats classes in groups rather than individually, is as effective as the traditional Chidamber and Kemerer metrics in identifying fault-prone classes.

1. INTRODUCTION

Since quantitative methods have significantly demonstrated their usefulness in other sciences, computer science researchers have worked hard to bring similar approaches to software development in the form of software metrics, a measure of some property of software code or its specifications. A plethora of OO design metrics has been proposed to help evaluate software design quality [4, 2, 6]. In order to demonstrate the usefulness of a metric during development of commercial applications, numerous empirical validations have also been performed and published within the literature. Many empirical studies on software metrics have linked quantitative design structures in OO designs to fault-proneness of classes [9, 10, 13], and most studies consistently use the Chidamber and Kemerer (CK) metric suite as a touchstone in predicting OO software quality [6]. Consequently, we use CK metrics as a comparison tool in this study.

In this paper, we present similar work from the empirical validation standpoint, but with an innovative suite of metrics based on a *Class Abstraction that uses a Taxonomy for OO classes* (CAT) to capture aspects of software complexity through combinations of class characteristics. The advantage of our approach is that it significantly reduces the analysis overhead by grouping classes based on these characteristics. A potential disadvantage of our approach is that important information may be lost by using taxonomy entries rather than full analysis data.

We empirically validate the ability of the CAT metrics to predict fault-prone classes using fault data for six versions of the Java-based open-source Eclipse Integrated Development Environment (IDE) [17]. We conclude that our proposed metric suite and CK metrics both produce statistical models that effectively identify fault-prone classes. Evaluating generated prediction models across six Eclipse IDE versions suggests that models generated from our proposed metrics are at least as effective in assessing the quality of OO classes as their CK metrics model counterpart.

The reminder of the paper is organized as follows. The class abstraction using the taxonomy of OO classes is reviewed in Section 2. Details of the empirical study are given in Section 3 and the prediction models are described and analysed in Section 4. Finally, we present the related work in Section 5 and conclude in Section 6.

2. CLASS ABSTRACTION

The class abstraction technique used in this paper is the taxonomy of OO classes previously described by Clarke *et al.* [7]; in this section we summarise its main elements.

- The **taxonomy** of OO classes is defined in terms of class characteristics.
- The **class characteristics** for a given class C are defined as the properties of the features (attributes and methods) in C and the dependencies C has with other types (built-in and user-defined) in the implementation.
- The **properties** of the features in C describe how criteria such as types, accessibility, shared class features, polymorphism, dynamic binding, deferred features, exception handling, and concurrency are represented in the attributes and routines of C.
- The **dependencies** C has with other types are realized through declarations and definitions of C's features, and C's role in an inheritance hierarchy.

The artifact generated when a class is cataloged using the taxonomy is referred to as a *cataloged entry*.

SAC'11 March 21-25, 2011, TaiChung, Taiwan.

Table 1: Descriptors used in a cataloged entry for a Java class.

Descriptors		
Nomenclature	*Attributes*	*Routines*
(Public)	(Transient)	(Final)
(Final)	(Volatile)	(Native)
(Has-Nested)	New	(Generic)
(Has-Inner)	Recursive	New
(Interface)	Concurrent	Recursive
(Implements)	Polymorphic	Redefined
(Serializable)	Private	Concurrent
Generic	Protected	Synchronized
Concurrent	Public	Exception-R
Abstract	Constant	Exception-H
Inheritance-free	Static	Has-Polymorphic
Parent	-	Non-Virtual
External Child	-	Virtual
Internal Child	-	Deferred
-	-	Private
-	-	Protected
-	-	Public
-	-	Static

A **cataloged entry** is a 5-tuple (*C, N, A, R, F*), where:

- *C* is the fully qualified name of the class.
- *N*, the *Nomenclature Component*, represents a group (or taxon) in the taxonomy and contains a single entry.
- *A*, the *Attributes Component*, is a list of entries representing the different groups of attributes.
- *R*, the *Routines Component*, is a list of entries representing the different groups of routines.
- *F*, the *Feature Classification Component*, is a list of entries summarizing the inherited features.

Each group listed in the components *C, N, A, R,* or *F* is referred as a *component entry* and consists of two parts: (1) the modifier that identifies the characteristics of the entry, and (2) a list of type families that identifies the types associated with that entry. A modifier consists of a list of descriptors (core and add-on) representing the class characteristics. The core descriptors represent class characteristics found in most OO languages and the add-on descriptors represent characteristics peculiar to a given language. Table 1 shows the descriptors used in the modifier part of the component entries in the Nomenclature, Attributes and Routines components, respectively. The add-on descriptors are shown in parentheses. The names of the descriptors were chosen to symbolize the characteristic they represent. For example, the add-on descriptor *Final*, Column 1 Row 2, indicates that the definition of the class is complete and no subclasses are allowed.

A more detailed explanation of the descriptors and type families are provided in [7], and the approach has been implemented using a Taxonomy Tool for the Object-Oriented Language Java, *TaxTOOLJ* [1]. The significant saving in terms of the analysis overhead for Java classes has been demonstrated in a study of 22 Java applications, which showed that of all the possible groups representing the combinations of class characteristics only a small percentage were used [7]. For example, 19,720 classes in Eclipse 3.1.1 were cataloged and of the 20,992 possible groups for Java 1.4 classes, only 401 groups were used. Similarly, 17,343 classes in the Java Development Kit 1.5.0.5 were cataloged and only 498 of the possible 3,497,840 groups were used.

Table 2: Summary of fault counts per Eclipse version for the JDT subsystem (package root *org.eclipse.jdt*).

Eclipse Version	# of Packages	# of Classes	# of Classes with Faults	# of Faults
2.0	126	2,218	719	2,307
2.1	133	2,679	767	1,964
3.0	158	3,389	1,204	4,001
3.1	169	3,904	1,369	4,339
3.2	205	4,574	1,611	4,085
3.3	219	4,877	1,289	2,733

3. EMPIRICAL STUDY

This section describes the conducted empirical study that explores the correlation between our CAT metric suite and fault-proneness. Within the scope of this paper, we test two research hypotheses:

- **Hypothesis 1.** *The proposed CAT metric suite can identify fault-prone classes in multiple, sequential releases of OO software systems written in Java.*
- **Hypothesis 2.** *The proposed CAT metric suite can produce a fault prediction model that is comparable to the prediction model derived from CK metrics.*

Target Applications

We conduct our experiment into the relationship between CAT metrics and fault-proneness by considering six versions of Java-based open-source Eclipse IDE [17]. We chose the Eclipse project to examine in this study since it is an example of a large-scale commerical software project that provides fault data for a significant series of releases. In this paper we investigate Eclipse's Java Development Tools (JDT) subsystem for the six official releases of Eclipse, versions 2.0, 2.1, 3.0, 3.1, 3.2, and 3.3, as shown in Table 2.

We chose the JDT subsystem as the candidate for this investigation mainly because there seems to be a significant amount of change in terms of new and changed classes in successive Eclipse versions. Class changes within the JDT subsystems play a considerable role in the development of the Eclipse platform from one version of Eclipse to the next. During the process of changing classes or introducing new classes we encounter greater likelihood of introducing faults than when software is neither being changed nor enhanced.

3.1 Metrics and Data Collection

The process of collecting data uses three sources. First we obtain CK metrics by using *Understand* software tool produced by Scientific Toolworks [14], and then we generate the CAT metrics using *TaxTOOLJ* [1]. Finally we generate fault statistics by accessing Bugzilla database.

Chidamber and Kemerer's (CK) Metrics

The *Understand* tool includes a set of static analysis tools that perform various metrics calculation. The following CK metrics were used in this study:

- *Weighted Methods per Class (WMC)*: Sum of complexities of local methods of a class.

- *Depth of Inheritance Tree (DIT)*: Maximum number of edges between a given class and a root class in an inheritance graph.
- *Number of Children (NOC)*: A count of the number of direct children of a given class.
- *Coupling between Objects (CBO)*: Counts other classes whose attributes or methods are used by the given class plus those that use the attributes or methods of the given class.
- *Response for a Class (RFC)*: A count of all local methods of a class plus all of the methods in other classes directly called by any of the methods in the class.
- *Lack of Cohesion of Methods (LCOM)*: Number of disjoint sets of local methods, any two methods in the same set share at least one local variable.

CAT Metrics

We consider diversity of methods, different types of object coupling, and exception handling when measuring the robustness of software systems. We therefore define the following initial set of CAT metrics:

- *Number of Distinct Non-Recursive Routine Component Entries (RCE)*: A count of the number of distinct component entries per class that are not inherited from parent classes. Only local methods of a class are considered. Component entries with descriptors *Recursive* are excluded from this metric.
- *Variability (VAR)*: A ratio of the number of distinct Non-Recursive Routine Component Entries (RCE) and total number of local methods for a given class. The metric has a range from 0 (no local methods present) to 1 (every local method within class belongs to a different routine component entry). Lower value of the metric indicates less variability in different routine component entries for a single class.
- *Number of Methods that Handle Exceptions (NEH)*: A count of the methods that explicitly handle exceptions; thus, they contain *try* and *catch* blocks within their respective bodies.
- *Coupling between User Defined Objects (CUS)*: Count of all attribute references to objects of other user-defined data types and all local methods that reference objects of other user-defined data types.
- *Coupling between Library Objects (CLS)*: Count of all attribute references to objects of library data types and all local methods that reference objects of other library data types.

Fault Data

We collected the fault data for the six official releases of the Eclipse project from the Bugzilla database. Initially, we accessed class modification reports obtained through Concurrent Versions System (CVS) Change Log plugin for Eclipse which provides a summary of CVS log entries [11]. Each entry consisted of reported fault identification number (ID) followed by the list of *.java* file names that were changed due to that specific fault.

We conducted a manual inspection of these change logs to identify the keywords that indicate fix revisions, and mapped these to the corresponding classes. Since any specific fault (identified by its ID) was possibly distributed over several *.java* files, the total count of all faults we consider in this study exceeds the actual number of individual fault IDs, as discussed in Section 4.4.

Every fault id reference number obtained was additionally checked for its severity classification within the Bugzilla database. Severities classified as *Trivial* (cosmetic problems like misspelled words or misaligned text) or *Enhancement* (request for enhancement) were excluded from the study. Only severities classified as *Blocker* (blocks development and/or testing work), *Critical* (crashes, loss of data, severe memory leak), *Major* (major loss of function), *Normal* (regular issue, some loss of functionality under specific circumstances), and *Minor* (minor loss of function, or other problem where easy workaround is present) were counted. Due to a lack of fault distribution within these severity classification levels, we do not consider exploring the link between individual metrics and fault severity levels.

4. PREDICTION MODELS

In this study we develop models using CK metrics and CAT metrics to predict faults. Because of the lack of variability in the response variable we chose to use logistic regression instead of the traditional linear regression technique. We develop binary logistic regression models with the SPSS statistical analysis software tool using the forward stepwise regression method. Logistic regression does not assume linearity of relationship between the independent and dependent variable nor does it require normally distributed variables. Binary logistic regression has been shown to provide good models for fault-proneness prediction in previous studies [8].

4.1 Collinearity (VIF) Analysis

In order to identify which metrics to use in our multivariate binary logistic regression (MBLR) models, we perform a collinearity analysis to determine if there are any potential collinearity problems in the bivariate correlations between the metrics within their respective metric suite. When there is a perfect linear relationship among the predictors, the estimates for a regression model cannot be uniquely computed. The term collinearity implies that two variables are near perfect linear combinations of one another. The primary concern is that as the degree of multicollinearity increases, the regression model estimates of the coefficients become unstable and the standard errors for the coefficients can get wildly inflated. We compute the variance inflation factor (VIF) values for each predictor as a check for multicollinearity, which is the reciprocal of the *tolerance*. The *tolerance* is an indication of the percent of variance in the predictor that cannot be accounted for by the other predictors. The VIF values, as a rule of thumb, if greater than 3.0 may merit further investigation of potential regressors for multicollinearity problems.

We present the VIF analysis for the CK metrics suite in Table 3. The table shows VIF analysis with all potential regressors. The values indicate that there are no multicollinearity problems, since all values are within VIF threshold and objective values.

Our VIF analysis for the CAT metrics suite is presented in Table 4. Again, the first section shows the VIF analysis with all potential regressors. Collinearity problems are suspected with RCE and CUS since they exceed our VIF threshold value. After removing RCE from the model, a subsequent

Table 3: Collinearity Analysis for CK Metrics

VIF Values (Using all CK metrics)						
Eclipse	2.0	2.1	3.0	3.1	3.2	3.3
WMC	2.12	2.249	2.262	2.254	2.286	2.283
DIT	1.525	1.462	1.328	1.382	1.385	1.359
NOC	1.028	1.027	1.034	1.034	1.031	1.031
CBO	1.674	1.785	1.865	1.857	1.851	1.845
RFC	2.164	2.044	1.853	1.916	1.945	1.917
LCOM	1.159	1.184	1.185	1.163	1.162	1.156

Table 4: Collinearity Analysis for CAT Metrics

VIF Values (Using all CAT metrics)						
Eclipse	2.0	2.1	3.0	3.1	3.2	3.3
RCE	4.827	5.487	5.557	5.065	5.269	5.204
VAR	1.201	1.213	1.208	1.206	1.208	1.211
NEH	1.864	1.93	1.817	1.799	1.833	1.839
CUS	4.12	4.645	4.799	4.733	4.949	4.933
CLS	2.578	2.838	2.951	2.973	2.925	2.969
VIF Values (Removing RCE)						
Eclipse	2.0	2.1	3.0	3.1	3.2	3.3
VAR	1.144	1.161	1.145	1.164	1.171	1.176
NEH	1.837	1.903	1.812	1.787	1.816	1.825
CUS	2.305	2.513	2.553	2.695	2.794	2.792
CLS	2.067	2.268	2.408	2.483	2.553	2.505
VIF Values (Removing CUS)						
Eclipse	2.0	2.1	3.0	3.1	3.2	3.3
RCE	2.7	2.969	2.957	2.884	2.974	2.946
VAR	1.087	1.099	1.087	1.091	1.099	1.1
NEH	1.777	1.851	1.739	1.731	1.767	1.763
CLS	2.578	2.837	2.948	2.929	2.973	2.92

VIF analysis shows that the remaining variables are within VIF threshold and objective values. The removal of CUS also shows similar results, and thus can be successfully used as another model configuration.

Based on these results, we developed and explored the performance of one model for CK metrics and two models for our CAT metrics:

- CK model: WMC, DIT, NOC, CBO, RFC, LCOM
- CAT metrics model 1: VAR, NEH, CUS, CLS
- CAT metrics model 2: RCE, VAR, NEH, CLS

4.2 The MBLR Model

The Eclipse version 2.0 MBLR results of the CK model are presented in Table 5. All investigated CK metrics are significant regressors in the model (p-value<0.05) for Eclipse version 2.0. However, that was not the case for every Eclipse version, since the level of significance of some of the regressors varies from version to version. For example, even though NOC metric was not a significant regressor for Eclipse versions 2.1, 3.0, and 3.3, it was significant for versions 2.0, 3.1, and 3.2, and showed a good fit of data for versions 3.1 and 3.2 as observed during univariate binary logistic regression (UBLR). Thus we intend to use multivariate model throughout all the versions of the Eclipse software with all six CK metrics included. Because of the space constraints, we do not show the CK metrics suite model MBLR results for Eclipse versions 2.1 through 3.3.

Table 5: Eclipse Version 2.0 Multivariate Logistic Regression for CK Metrics

MBLR Results for CK Metrics Suite Model							
	Const.	WMC	DIT	NOC	CBO	RFC	LCOM
Coeff.	-2.178	0.019	0.183	-0.062	0.081	-0.007	0.011
SE Coeff.	0.130	0.005	0.046	0.030	0.008	0.002	0.002
odds ratio	-	1.020	1.201	0.940	1.084	0.993	1.011
p-value	0.000	0.000	0.000	0.039	0.000	0.000	0.000

Table 6: Eclipse Version 2.0 Multivariate Logistic Regression for CAT Metrics

MBLR Results for CAT Metrics Suite Model #1					
	Constant	VAR	NEH	CUS	CLS
Coeff.	-1.931	-	0.437	0.103	0.062
SE Coeff.	0.084	-	0.054	0.011	0.015
odds ratio	-	-	0.084	1.109	1.064
p-value	0.000	0.174	0.000	0.000	0.000
MBLR Results for CAT Metrics Suite Model #2					
	Constant	RCE	VAR	NEH	CLS
Coeff.	-2.161	0.171	-	0.420	0.044
SE Coeff.	0.103	0.021	-	0.054	0.016
odds ratio	-	1.186	-	1.522	1.045
p-value	0.000	0.000	0.228	0.000	0.005

The Eclipse version 2.0 MBLR results of the CAT models are presented in Table 6. All investigated CAT metrics are significant regressors in the model (p-value<0.05) except for the VAR metric. Unlike CK NOC regressor within the CK model, MBLR results for the VAR metric within both CAT metrics suite models for all versions of Eclipse showed it not to be a significant (p-value>0.05) regressor. Additionally, VAR metrics did not produced a good fit of data for a single version of Eclipse software during UBLR. We have therefore decided to remove the VAR metric from both our models. Again, because of the space constraints, we do not show the CAT metrics suite model MBLR results for Eclipse versions 2.1 through 3.3.

4.3 Evaluation of Models

To validate our CK and CAT metric suite models, we develop MBLR models with Eclipse version n data, and validate them with Eclipse version $n+1$ data. We choose our CAT metric suite model #1 for Eclipse version 2.0 as an example. Given the MBLR data from CAT metric suite model #1 for Eclipse version 2.0 in Table 6, we develop the general MBLR model applicable for Eclipse version 2.1 as:

$$\ln(\frac{p}{1-p}) = -1.931 + 0.437_{\text{NEH}} + 0.103_{\text{CUS}} + 0.062_{\text{CLS}}$$

Therefore, we calculate the probability p that the fault in a class from Eclipse version 2.1 is present as follows:

$$p = \frac{e^{(-1.931+0.437_{\text{NEH}}+0.103_{\text{CUS}}+0.062_{\text{CLS}})}}{1+e^{(-1.931+0.437_{\text{NEH}}+0.103_{\text{CUS}}+0.062_{\text{CLS}})}}$$

We evaluate the performance of the model expressed in percentage of classes whose fault probability values correctly classified them as fault-free (p<0.5) or fault-prone (p≥0.5).

Table 7 shows in detail the overall performance of all three models across all versions of Eclipse. Each of the five subtables of data in Table 7 measures the ability of the pre-

Table 7: Detailed results of prediction capability for five versions of Eclipse. In each case the model developed from the first Eclipse version is used to predict faults in the second.

Eclipse Version:	2.0 to 2.1			2.1 to 3.0			3.0 to 3.1			3.1 to 3.2			3.2 to 3.3		
Metrics Suite:	CAT1	CAT2	CK	CAT1	CAT2	CK	CAT1	CAT2	CK	CAT1	CAT2	CK	CAT1	CAT2	CK
Sensitivity (True Positive Rate):	49%	48%	44%	36%	35%	29%	46%	46%	47%	44%	43%	43%	52%	51%	53%
False Positive Rate:	7%	8%	10%	5%	4%	5%	11%	13%	13%	10%	9%	14%	13%	14%	14%
Accuracy:	80%	79%	76%	72%	72%	69%	72%	71%	71%	71%	71%	69%	76%	75%	76%
Specificity (True Negative Rate):	93%	92%	90%	95%	96%	95%	89%	87%	87%	91%	91%	86%	87%	86%	86%
Positive Predictive Value:	76%	73%	66%	83%	84%	79%	74%	71%	71%	76%	76%	69%	64%	61%	62%
Negative Predictive Value:	81%	80%	79%	69%	69%	67%	71%	71%	71%	70%	70%	69%	80%	80%	80%
False Discovery Rate:	24%	27%	35%	17%	16%	21%	26%	29%	29%	24%	24%	32%	37%	39%	38%

diction model derived from one version of Eclipse to predict faults in the next version, for each of the three metrics suites.

Based on the results obtained for the validation of Eclipse models, we accept our Hypothesis 1, that our CAT metrics can identify fault-prone classes in multiple, sequential releases of OO software systems written in Java. The predictive value of all the CAT metrics suite models was between 71% and 84% accuracy. We also accept Hypothesis 2, that the proposed CAT metric suite can produce a fault prediction model that is comparable to the prediction model derived from CK metrics. In fact, the CAT metrics models have actually outperformed their CK counterpart by classifying between 1% (Eclipse versions 3.1 and 3.3) and 4% (Eclipse version 2.1) additional classes correctly.

4.4 Threats to Validity

In Section 3 we discuss why we believe we chose the right application for this study. However, it is possible that quite different results might be achieved for a different application, or with applications written using a different programming language. Since there are no other empirical studies done on the proposed taxonomy metrics, it is difficult to validate our results.

In Section 3.1, we noted that the total count of all faults we consider in this study exceeds the actual number of individual fault IDs, because we count the number of faults as the number of individual accesses to *.java* files recorded in the analyzed change log. It is possible that access to the *.java* file was to fix either an inner class or a non-public class contained within it, but the fault was attributed to the actual public class having the same name as the *.java* file.

Emam *et al.* found that class size has a strong confounding effect on the validity of OO metrics [8], since the association between the investigated metrics and fault-proneness disappear after class size has been taken into account. We did not consider class size as an additional independent variable in this study.

5. RELATED WORK

Previous studies have empirically validated the association between OO metrics and fault-proneness, categorizing software modules into different groups based on the number of faults in a module. Typically they divide modules into two distinct categories: the modules that were faulty (contained one or more faults) and non-faulty modules (fault free) [3, 9, 10]. Basili *et al.* conducted experiments on student projects for which they collected fault data during acceptance testing [3]. Their results showed that CK metrics were statistically independent and all six CK metrics that were also used in our study were significantly associated with class fault-proneness. Gyimothy *et al.* also validated the CK metrics as significantly associated with class fault-proneness [10]. They investigated Mozilla version 1.6, an open source software, by collecting and investigating faults from the Bugzilla database. All metrics except NOC were significant predictors of class fault-proneness in their study.

Olague *et al.* empirically validated three sets of metric suites to predict fault-proneness of OO classes using highly iterative or agile software development process [13]. They studied the CK metrics, Abreu's Metrics for OO Design (MOOD) [5], and Bansiya and Davis' quality metrics for OO design (QMOOD) [2]. They used defect data for six versions of Rhino, an open source implementation of JavaScript written in Java, and concluded that the CK and QMOOD metric suites both produce statistical models that are effective in detecting fault-prone classes. In their study, MOOD metric suite was not effective in detecting fault-prone classes.

Subramanyam and Krishnan validated the association between WMC, CBO, and DIT metrics and the fault counts, rather than fault-proneness [15]. They analyzed around 400 C++ and 300 Java classes, concluding that CK metrics were significantly associated with faults counts, but they found that effectiveness of these metrics vary in the two programming languages investigated. While C++ classes found WMC, DIT, and interaction term (CBO*DIT) all significantly associated with faults counts, Java classes were significantly associated with faults through interaction term (CBO*DIT) only.

Szabo and Khoshgoftaar have also validated the association between traditional and OO metrics and fault-proneness of classes [16]. Unlike us, they classify software modules into three, rather than two (fault/no fault), distinct categories based on the number of faults: high (number of faults $>$ 19), medium (2 $<$ number of faults $\leq$ 19), and low (number of faults $\leq$ 2) risk groups. The results of the study showed that addition of OO measures enhanced the model significantly by reducing the overall misclassification rate.

Li and Shatnawi explored the link between the bad smells (a structure in the code that suggests opportunities for refactoring) and class fault probability in the evolution of OO systems [12]. They presented the empirical study that investigated the relationship between the bed smells and class fault probability in three fault-severity levels using three versions of an industrial-strength open source system Eclipse. They showed that some bad smells, identified by set of metrics and their threshold values, were positively associated with the class fault probability.

Zimmermann *et al.* conducted a study that maps fault

locations to the number of faults reported in the first six months before and after the release of the Eclipse, versions 2.0, 2.1, and 3.0 [19]. Unlike our study, they used both pre-release and post-release fault data at both the package-level and file-level, and suggested that while fewer of the fault prone files were correctly identified as fault-prone, there were hardly any false positives (i.e. when a file is classified as fault prone, its classification is most likely to be correct). Furthermore, they pose several follow-up questions concerning the origin of faults as well as the applicability of models for systems as they evolve over time. In our study we further investigate the applicability of models as they evolve over time by considering 6 successive versions of the Eclipse platform.

Zhang investigated the relationship between lines of code (LOC) and faults using two fault data sets, the Eclipse dataset and the NASA dataset [18]. Identically to Zimmermann *et al.* [19], Zhang used both pre-release and post-release fault data at both the package-level and file-level. The result of the study showed that larger modules tend to have more faults and a small number of the largest modules account for a large proportion of the faults. Zhang also showed that by using Weibull functions, a small percentage of the largest modules can predict total number of faults at both the package and file levels.

6. CONCLUDING REMARKS

We proposed an innovative metrics suite based on a class abstraction technique that uses a taxonomy for OO classes (CAT) to measures quantitative properties of OO classes within software system using combinations of class characteristics. The CAT metrics include refined object coupling and exception handling measurements. We conducted a statistical analysis of the widely used CK class metrics and CAT metrics using six versions of an industrial-strength open source system Eclipse.

The results of this case study indicate that both CK and CAT metrics are effective in developing quality statistical models to predict faults in multiple, sequential releases of OO software systems written in Java. Additionally, the results indicate that despite their higher level of abstraction, CAT metrics have the ability to produce a comparable fault prediction models when compared to the models derived from CK metrics.

In future work we propose to expand our models to include fault data from other applications used in some of the studies described in Section 5.

7. REFERENCES

[1] D. Babich, K. Chiu, and P. J. Clarke. TaxTOOLJ: A tool to catalog Java classes. In *18th International Conference on Software Engineering and Knowledge Engineering*, pages 375 – 380, July 2006.

[2] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. on Softw. Eng.*, 28(1):4–17, 2002.

[3] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. on Softw. Eng.*, 22(10):751–761, 1996.

[4] F. Brito e Abreu and W. Melo. Evaluating the impact of object-oriented design on software quality. In *International Symposium on Software Metrics*, pages 90–99, Berlin, Germany, 1996.

[5] F. Brito e Abreu, G. Pereira, and P. Sousa. A coupling-guided cluster analysis approach to reengineer the modularity of object-oriented systems. In *Conference on Software Maintenance and Reengineering*, pages 13–22, Zurich, Switzerland, 2000.

[6] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. on Softw. Eng.*, 20(6):476–493, 1994.

[7] P. J. Clarke, D. Babich, T. M. King, and B. M. Golam Kibria. Analyzing clusters of class characteristics in oo applications. *J. Syst. Softw.*, 81(12):2269–2286, 2008.

[8] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Trans. on Softw. Eng.*, 27(7):630–650, 2001.

[9] K. E. Emam, W. Melo, and J. C. Machado. The prediction of faulty classes using object-oriented design metrics. *J. Syst. Softw.*, 56(1):63–75, 2001.

[10] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. on Softw. Eng.*, 31(10):897–910, 2005.

[11] J. C. Landers and M. Spiegel. CVS change log for Eclipse. `http://sourceforge.net/projects/cvschangelog/`, 2004.

[12] W. Li and R. Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *J. Syst. Softw.*, 80(7):1120–1128, 2007.

[13] H. M. Olague, L. H. Etzkorn, S. Gholston, and S. Quattlebaum. Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. *IEEE Trans. on Softw. Eng.*, 33(6):402–419, 2007.

[14] Scientific Toolworks Inc. Understand. `http://www.scitools.com/products/understand/`, 2004.

[15] R. Subramanyam and M. S. Krishnan. Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects. *IEEE Trans. on Softw. Eng.*, 29(4):297–310, 2003.

[16] R. M. Szabo and T. M. Khoshgoftaar. An assessment of software quality in a C++ environment. In *International Symposium on Software Reliability Engineering*, pages 240–249, Toulouse, France, 1995.

[17] The Eclipse Foundation. Eclipse. `http://www.eclipse.org/`, 2010.

[18] H. Zhang. An investigation of the relationships between lines of code and defects. In *International Conference on Software Maintenance*, pages 274–283, Edmonton, Canada, 2009.

[19] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for Eclipse. In *International Workshop on Predictor Models in Software Engineering*, page 9, Minneapolis, MN, 2007.